2020 年 7 月 19 日

ネットワークプログラミング

TCP/UDP 通信とブロードキャスト（設計書）

学籍番号:18122508　情報工学課程　渡邉紘矢

1　共通して使用する定数値

- サーバー検索時の返り値

  プログラムの起動時に HERO パケットを送信 し、返事があるかどうかでサーバーが既に起動しているか、していないかを判断する。この結果の返り値として用い、主に可読性のために使用を検討している。

ソースコード 1　サーバー検索時の返り値

```
1  #define SERVER_NOT_EXIST 0
2  #define SERVER_EXIST 1
```

- パケットのタイプ

  取得するパケットの種類は全部で 6 種類である。どの種類のパケットを作成するのか、どのパケットを受信したのかを判断するために用いる。

ソースコード 2　パケットのタイプ

```
1  #define HELLO 1
2  #define HERE 2
3  #define JOIN 3
4  #define POST 4
5  #define MESSAGE 5
6  #define QUIT 6
```

- UDP 通信時に用いるタイムアウトの時間

  サーバー検索には UDP を用いる。UDP はコネクションレスな通信であるため、相手に必ずパケットが届くことが保証されておらず、送信すれば返ってくるはずの返事が返ってこない場合がある。そのための、タイムアウトの時間を設定するために用いる。単位は秒である。

ソースコード 3　UDP 通信時に用いるタイムアウトの時間

```
1  #define TIMEOUT_SEC 1
```

- 各データのサイズ

  仕様書に従い、プログラム内で適切にバッファやデータ処理を行うために、各データのサイズを定義しておく。

ソースコード 4　各データのサイズ

```
1  /* メッセージ全体のサイズは512バイト */
2  #define MSGBUF_SIZE 512
3  /* パケット内においてデータが入る部分は488バイト */
4  #define MSGDATA_SIZE 488
5  /* ユーザ名格納用バッファサイズ */
6  #define USERNAME_LEN 15
```

- 文字色と背景色のペアを表す定数値

  今回は、ncurses ライブラリを用いて、端末制御を行う。ncurses では、文字色と背景色のペアを定数値に登録し、その値を用いて色のペアを呼び出すことになる。その色を明確に呼び出せるように用いる。

ソースコード 5　文字色と背景色のペアを表す定数値

```
1  #define COL_BLK_WHT 1
2  #define COL_GRN_WHT 2
3  #define COL_CYN_WHT 3
```

2　共通して使用する構造体

- ユーザの情報を格納する構造体

　ユーザ情報を格納するために、線形リストの構造を利用する。ユーザはソケット番号を用いて一意に特定することが可能であり、ソケット番号を用いてユーザ名を特定できるようにする。また、ユーザが途中で抜けることがあり、ユーザの情報を削除する際に、線形リストであれば 1 箇所のポインタを設定しなおすことで実現できる。

ソースコード 6　ユーザの情報を格納する構造体

```
1  typedef struct _imember {
2      char username[USERNAME_LEN]; /* ユーザ名 */
3      int sock;                    /* ソケット番号 */
4      struct _imember *next;       /* 次のユーザ */
5  } * member_t;
```

- パケットの構造

　送信するデータは、ヘッダとして 4 バイトの識別文字列を付加され、メッセージ本体とヘッダがスペースで区切られた文字列であるとする。この文字列が、以下のようなデータ構造になっているとして扱うと、受信したパケットのタイプ、メッセージ本体を扱いやすくできる。

ソースコード 7　パケットの構造

```
1  typedef struct _idobata {
2      char header[4]; /* パケットのヘッダ部(4バイト) */
3      char sep;       /* セパレータ(空白、またはゼロ) */
4      char data[MSGDATA_SIZE]; /* データ部分(メッセージ本体) */
5  } ido_packet_t;
```

3　ウィンドウ作成用ソースコード (window.c)

window.c には、以下のような定数値と関数を用意する。ここには、プロトタイプ宣言のみ記述することにする。

ソースコード 8　window.c の実装

```
1  #define SUBWIN_LINES 5 /* サブウィンドウの行数 */
2
3  static void init_color_pair(); /* Color pairの定義 */
4
5  void create_window(WINDOW **win_main, WINDOW **win_sub); /* WINDOWを作成 */
```

- #define SUBWIN_LINES 5

　メッセージを表示するメインウィンドウと、自分が送信したいメッセージを入力するサブウィンドウを用意する。そのサブウィンドウの行数を指定する。

- static void init_color_pair()

　以下の 3 種類の背景色と文字色のペアを登録する。

  - 背景色:白　文字色:黒
  - 背景色:白　文字色:緑
  - 背景色:白　文字色:水色

　引数:　なし
　返り値:　なし

- void create_window(WINDOW **win_main, WINDOW **win_sub)

　メインウィンドウとサブウィンドウを作成する。以下の初期設定が完了したウィンドウがセットされる。

- 日本語を表示出来るようにする
- 色を変更出来るようにする
- 使用する色を定義する
- 背景色を白、文字色を黒にする
- スクロールを許可する

引数:

- win_main: 作成されたメインウィンドウ。受信したメッセージを表示するウィンドウになる。
- win_sub: 作成されたサブウィンドウ。ユーザのキーボード入力を表示するウィンドウになる。

返り値: なし

4 パケットの解析 + 作成用ソースコード (packet.c)

packet.c には、以下のような関数を用意する。

ソースコード 9 packet.c の実装

```
1  /*
2   * パケットの種類=type のパケットを作成する
3   * パケットのデータは 内部的なバッファ(Buffer)に作成される
4   */
5  void create_packet(char *Buffer, unsigned int type, char *message);
6
7  /* パケットのタイプを解析する */
8  unsigned int analyze_header(char *header);
```

- void create_packet(char *Buffer, unsigned int type, char *message)

  パケットの種類と、送信したいメッセージ本体からパケットを作成する。

  引数:

  - Buffer: 作成されたパケット。セットされたデータを TCP や UDP 通信で送信する。
  - type: 作成したいパケットの種類
  - message: 送信したいメッセージ本体

  返り値: なし

- unsigned int analyze_header(char *header)

  パケットの種類を調べ、種類を表す整数値を返す。

  引数:

  - header: 受信したパケットのヘッダ部分

  返り値: パケットの種類を表す整数値

5 ユーザ情報へアクセスするための線形リストの実装 (linear_lists.c)

packet.c には、以下のような変数と関数を用意する。

ソースコード 10 linear_lists.c の実装

```
1  static member_t head = NULL; /* 線形リストの先頭 */
2  static member_t tail = NULL; /* 線形リストの末尾 */
3
4  /* 新しくメンバーを生成する関数 */
5  member_t create_member();
6  /* 線形リストの先頭を取得する */
7  member_t get_head_from_list();
8  /* 新しいメンバーを線形リストの末尾に追加する */
9  void add_user_to_list(char *username, int sock);
10 /* sockを指定して、そのメンバーを削除する */
11 void delete_user_from_list(int sock);
```

- `static member_t head = NULL`
  線形リストの先頭を指すポインタ。線形リストにデータが無い時は NULL 値になる。

- `static member_t tail = NULL`
  線形リストの末尾を指すポインタ。線形リストにデータが無い時は NULL 値になる。

- `static member_t create_member()`
  新たなユーザのデータを格納するために、動的に構造体分のメモリを確保し、そのポインタを返す。
  引数: なし
  返り値: 1 人分のデータを格納する構造体へのポインタ

- `member_t get_head_from_list()`
  線形リストの先頭を指すポインタを返す。
  引数: なし
  返り値: 線形リストの先頭を指すポインタ

- `void add_user_to_list(char *username, int sock)`
  ユーザの情報として、ユーザ名とソケット番号を格納し、線形リストの末尾に追加する。
  引数:
  - username: ユーザの名前
  - sock: そのユーザと TCP 通信を行うためのソケット番号

  返り値: なし

- `void delete_user_from_list(int sock)`
  ソケット番号を用いて、ユーザを特定し、そのユーザ情報を線形リストから削除する。
  引数:
  - sock: そのユーザと TCP 通信を行うためのソケット番号

  返り値: なし

6 共通で使用する関数のソースコード (idobata_common.c)

idobata_common.c には、以下のような変数と関数を用意する。

ソースコード 11 idobata_common.c の実装

```
1  char server_addr[20]; /* サーバーアドレス */
2  char user_name[USERNAME_LEN]; /* ユーザー名 */
3  /* サーバーを検索する */
4  int search_server(int port_number);
5  /* サーバーアドレスを格納する */
6  void set_server_addr(char *addr);
7  /* 格納されているサーバーアドレスを取得する */
8  void get_server_addr(char *out);
9  /* ユーザー名を格納する */
10 void set_user_name(char *name);
11 /* 格納されているユーザ名を取得する */
12 void get_user_name(char *out);
13 /* 自分が送信したメッセージを表示する */
14 void show_your_msg(WINDOW *win, char *buf);
15 /* 他の人の送信したメッセージを表示する */
16 void show_others_msg(WINDOW *win, char *buf);
17 /* 日本語の出現回数をカウントする関数 */
18 int cnt_jp(char *str);
```

- `char server_addr[20]`

サーバーを検索した結果、サーバーから返事があった時、そのサーバーの IP アドレスを格納するための変数。TCP 通信においてコネクションを確立する際に用いる。

- `char user_name[USERNAME_LEN]`
  JOIN パケットを送信する際にユーザ名をメッセージ本体に含めるため、変数にセットしておく。

- `int search_server(int port_number)`
  ブロードキャストを行い、ネットワーク上にサーバーが存在するか調べる。タイムアウトを 3 回するまでパケットを再送する。3 回の送信でサーバーが見つからなかった場合は 0 を返し、見つかった場合は即座に 1 を返す。
  引数:
  - port_number: サーバーが通信を待ち受けているポート番号

  返り値: サーバーが見つかった時は 1、見つからなかった時は 0 を返す

- `void set_server_addr(char *addr)`
  サーバーの IP アドレスを変数に格納する。
  引数:
  - addr: サーバーの IP アドレス

  返り値: なし

- `void get_server_addr(char *out)`
  記憶しているサーバーの IP アドレスを引数に指定された配列にセットする。
  引数:
  - out: サーバーの IP アドレスがセットされる配列

  返り値: なし

- `void set_user_name(char *name)`
  ユーザ名を変数に格納する。
  引数:
  - name: ユーザ名

  返り値: なし

- `void get_user_name(char *out)`
  記憶しているユーザ名を引数に指定された配列にセットする。
  引数:
  - out: ユーザ名がセットされる配列

  返り値: なし

- `void show_your_msg(WINDOW *win, char *buf)`
  背景色を白、文字色を緑色として、右端揃えで文字列を表示する。
  引数:
  - win: 文字列を表示するウィンドウ
  - buf: 表示したい文字列

  返り値: なし

- `void show_others_msg(WINDOW *win, char *buf)`
  背景色を白、文字色を水色として、左端揃えで文字列を表示する。
  引数:
  - win: 文字列を表示するウィンドウ
  - buf: 表示したい文字列

  返り値: なし

- `int cnt_jp(char *str)`
  受け取った文字列の中に含まれている日本語の文字数を数え、その文字数を返す。
  引数:
  - str: 調査対象となる文字列

  返り値: 含まれている日本語の文字数

7 クライアント用関数のソースコード (idobata_client.c)

idobata_client.c には、以下のような変数と関数を用意する。

ソースコード 12 idobata_client.c の実装

```
1 static WINDOW *win_main, *win_sub; /* ウィンドウ */
2 /* クライアントを初期化する */
3 static void init(int port_number, int *sock);
4 /* クライアントとして実行する */
5 void idobata_client(int port_number);
6 /* サーバーに参加する */
7 int join_server(int port_number);
8 /* キーボード入力されたメッセージを送信する */
9 void send_msg_from_keyboard(int sock, char *p_buf);
```

- `static WINDOW *win_main, *win_sub`
  メッセージを表示するメインウィンドウと、ユーザのキーボード入力を表示するサブウィンドウ

- `static void init(int port_number, int *sock)`
  以下の順序でクライアントの初期設定を行う。
  (1) ウィンドウを作成する
  (2) サーバーと TCP 通信のコネクションを確立させる
  (3) サーバーに JOIN パケットを送信する
  (4) サーバーとの通信に用いるソケット番号を sock 変数にセットする
  引数:
  - port_number: サーバーのポート番号
  - sock: サーバーと通信するためのソケット番号

  返り値: なし

- `int join_server(int port_number)`
  サーバーと TCP 通信のコネクションを確立させ、サーバーに JOIN パケットを送信する。今後もサーバーと通信を行うので、そのソケット番号を返す。
  引数:
  - port_number: サーバーのポート番号

  返り値: サーバーと通信するためのソケット番号

- `void send_msg_from_keyboard(int sock, char *p_buf)`

キーボード入力を受け取った文字列を用いて POST パケットを作成し、サーバーに送信する。

引数:

- sock: サーバーとの通信に用いるソケット番号
- p_buf: キーボード入力を受け取った文字列

返り値: なし

8 サーバー用関数のソースコード (idobata_server.c)

idobata_client.c には、以下のような定数値、変数と関数を用意する。

ソースコード 13 idobata_server.c の実装

```
1  #define FROM_SERVER -1 /* メッセージ送信者がサーバーの時にsockに渡す値 */
2
3  static WINDOW *win_main, *win_sub; /* Window */
4  fd_set mask, readfds; /* Selectで監視するためのマスクと結果 */
5  static int Max_sd = 0; /* selectで監視する最大値 */
6
7  /* サーバーを初期化する */
8  static void init(int port_number, int *server_udp_sock, int *server_tcp_sock, int *client_tcp_sock);
9  /* サーバーとして実行する */
10 void idobata_server(int port_number);
11 /* UDPパケットを受信したときの処理 */
12 static void recv_udp_packet(int udp_sock);
13 /* クライアントからパケットを受信したときの処理 */
14 static void recv_msg_from_client();
15 /* ユーザの登録情報を削除する */
16 static void delete_user(char *user_name, int sock);
17 /* ユーザ名前を登録する */
18 static void register_username(member_t user, ido_packet_t *packet);
19 /* メッセージを転送する */
20 static void transfer_message(char *message, char *from_user_name, int from_sock);
21 /* selectで監視する値を更新する */
22 static void setMax_sd(int num);
```

- `#define FROM_SERVER -1`

  サーバーがメッセージを転送する際には、メッセージ本体の先頭にユーザ名を付加してから送信する。そのため、ユーザ名を取得するためにソケット番号と一致するまで線形リストを走査する必要がある。この時、サーバーの情報は線形リストに含まれないため、転送ではなく、サーバーから直接メッセージを送信したい場合はソケット番号を-1 とすることで上手く処理する。

- `static WINDOW *win_main, *win_sub`

  メッセージを表示するメインウィンドウと、ユーザのキーボード入力を表示するサブウィンドウ

- `fd_set mask, readfds`

  select を用いて通信やキーボード入力を監視するためのマスク (mask) と監視状況をセットする (readfds) 変数

- `static int Max_sd = 0`

  select を用いて監視する値の最大値がセットされる

- `static void init(int port_number, int *server_udp_sock, int *server_tcp_sock, int *client_tcp_sock)`

  サーバーの初期設定を行う。サーバーは各クライアントの接続を受け付け、各クライアントにメッセージを転送する役割を持つ。また、サーバーはサーバーとしてだけでなく、1 つのクライアントとしても実行されるようになっている。それぞれの通信に用いるソケット番号が引数に指定された変数にセットされる。よってサーバーは以下の順序で初期化を行う。

  (1) ウィンドウを作製する

(2) UDP サーバーを初期化する
(3) TCP サーバーを初期化する
(4) クライアントを初期化し、サーバーに接続する
(5) 各ソケット番号を変数にセットする

引数:

- port_number: サーバーとして通信を待ち受けるポート番号
- server_udp_sock: UDP 通信を行うためのソケット番号
- server_tcp_sock: クライアントがコネクション確立のために通信してくるのを待ち受けるソケット番号
- client_tcp_sock: サーバーは同時にクライアントとしても実行されている。そのクライアントがサーバーと通信を行うためのソケット番号

返り値: なし

- `void idobata_server(int port_number)`
  指定されたポート番号でサーバーを実行する。また同時に、そのサーバーに接続するクライアントも実行することで、サーバー兼クライアントとして振る舞う。
  引数:
  - port_number: サーバーとして通信を待ち受けるポート番号

  返り値: なし

- `static void recv_udp_packet(int udp_sock)`
  UDP パケットを受信し、パケットの情報をもとにサーバーを検索しているクライアントに対して、HERE パケットを送信する。
  引数:
  - udp_sock: UDP 通信に用いるためのソケット番号

  返り値: なし

- `static void recv_msg_from_client()`
  クライアントからデータが届いているか確認し、受信できる相手からデータを受信する。届いたパケットの種類や内容に応じて以下のような分岐処理を行う。
  - クライアントが切断したならユーザの登録情報を削除する
  - 名前が登録されていない場合、JOIN パケットを待ち、登録させる
  - パケットの種類が POST だったらメッセージを転送する
  - パケットの種類が QUIT だったら、コネクションを切断し、ユーザの登録情報を削除する

  引数: なし
  返り値: なし

- `static void register_username(member_t user, ido_packet_t *packet)`
  パケットの種類が JOIN だったら、パケットのメッセージ本体をユーザ名として登録する。また、新たにクライアントが参加したことを全クライアントに通知する。
  引数:
  - user: ユーザ情報を格納している構造体へのポインタ
  - packet: 受信したパケットの構造体へのポインタ

  返り値: なし

- `static void delete_user(char *user_name, int sock)`
  クライアントがサーバーから切断したことを全クライアントに通知する。そのユーザのソケット番号を select で監視する対象外とし、ユーザの情報を線形リストから削除する。
  引数:
  - user_name: 削除するユーザ情報のユーザ名
  - sock: 削除するユーザと通信するためのソケット番号

  返り値: なし

- `static void transfer_message(char *message, char *from_user_name, int from_sock)`
  転送するメッセージ本体の先頭にユーザ名を付加し、MESSAGE パケットを作成する。もとの送信者であるクライアント以外にそのパケットを送信する。
  引数:
  - message: 転送するメッセージ本体
  - from_user_name: メッセージをサーバに送信してきたクライアントのユーザ名
  - from_sock: メッセージをサーバに送信してきたクライアントのソケット番号

  返り値: なし

- `static void setMax_sd(int num)`
  select を実行する際に、監視する値の最大値 +1 を指定する必要がある。監視する最大値はクライアントとコネクションを確立するごとに変化することが考えられ、この関数を用いて常に最大値がセットされるようにする。
  引数:
  - num: select で監視する値

  返り値: なし

9 井戸端会議エントリポイントのソースコード (idobata.c)

`idobata.c` には、以下のような定数値、変数と関数を用意する。

ソースコード 14 idobata.c の実装

```
1 #define DEFAULT_PORT 50001 /* ポート番号既定値 */
2 /* 実行時引数 */
3 extern char *optarg;
4 extern int optind, opterr, optopt;
5 /* エントリポイント */
6 int main(int argc, char *argv[]);
7 /* ヘルプ表示 */
8 void help_message(char *script_name);
```

- `#define DEFAULT_PORT 50001`
  デフォルトのポート番号を指定する。実行時引数にポート番号が指定されなければポート番号として 50001 を用いる。

- `extern char *optarg`
- `extern int optind, opterr, optopt`
  実行時引数にオプション指定を行うために予め宣言しておく変数

- `int main(int argc, char *argv[])`
  エントリポイント。以下の順序でクライアント、もしくはサーバーの動作へ分岐する。

(1) 実行時引数のオプションを解析し、各値をセットする
(2) サーバがネットワーク内に存在するか検索する
(3) サーバが存在すれば、クライアントとして実行する。いなければ、サーバーとして実行する。

- `void help_message(char *script_name)`
  プログラムの使い方を表示する。
  引数:
  - script_name: プログラム実行時の実行ファイル名

  返り値: なし

10 ヘッダファイル (idobata.h)

以上をまとめると、ヘッダファイルに記述する内容は以下のようになる。

ソースコード 15 idobata.h

```c
#ifndef IDOBATA_H
#define IDOBATA_H

#include <arpa/inet.h>
#include <locale.h>
#include <ncurses.h>

/* サーバー検索時の返り値 */
#define SERVER_NOT_EXIST 0
#define SERVER_EXIST 1
/* Color pair */
#define COL_BLK_WHT 1
#define COL_GRN_WHT 2
#define COL_CYN_WHT 3

/* Packet Type */
#define HELLO 1
#define HERE 2
#define JOIN 3
#define POST 4
#define MESSAGE 5
#define QUIT 6

/* UDP通信でタイムアウトを使用する */
#define TIMEOUT_SEC 1

/* メッセージ全体のサイズは512バイト */
#define MSGBUF_SIZE 512
/* パケット内においてデータが入る部分は488バイト */
#define MSGDATA_SIZE 488
/* サーバ名格納用バッファサイズ */
#define USERNAME_LEN 15

typedef struct _imember {
    char username[USERNAME_LEN]; /* ユーザ名 */
    int sock;                    /* ソケット番号 */
    struct _imember *next;       /* 次のユーザ */
} * member_t;

/* パケットの構造 */
typedef struct _idobata {
    char header[4]; /* パケットのヘッダ部(4バイト) */
    char sep;       /* セパレータ(空白、またはゼロ) */
    char data[488]; /* データ部分(メッセージ本体) */
} ido_packet_t;

// ============================================
//  window.c
// ============================================

/* ウィンドウを作成する */
void create_window(WINDOW **win_main, WINDOW **win_sub);

// ============================================
//  idobata_common.c
// ============================================
int search_server(int port_number);
/* サーバーアドレスを格納する */
void set_server_addr(char *addr);

/* 格納されているサーバーアドレスを取得する */
void get_server_addr(char *out);
/* ユーザー名を格納する */
void set_user_name(char *name);
/* 格納されているユーザ名を取得する */
void get_user_name(char *out);
```

```
67  /* 自分が送信したメッセージを表示する */
68  void show_your_msg(WINDOW *win, char *buf);
69  /* 他の人の送信したメッセージを表示する */
70  void show_others_msg(WINDOW *win, char *buf);
71  /* 日本語の出現回数をカウントする関数 */
72  int cnt_jp(char *str);
73
74  // ============================================
75  //  packet.c
76  // ============================================
77  /*
78   * パケットの種類=type のパケットを作成する
79   * パケットのデータは 内部的なバッファ(Buffer)に作成される
80   */
81  void create_packet(char *Buffer, unsigned int type, char *message);
82  /* パケットのタイプを解析する */
83  unsigned int analyze_header(char *header);
84
85  // ============================================
86  //  idobata_server.c
87  // ============================================
88  /* サーバーを起動する */
89  void idobata_server(int port_number);
90
91  // ============================================
92  //  idobata_client.c
93  // ============================================
94  /* クライアントを起動する */
95  void idobata_client(int port_number);
96  /* サーバーに参加する */
97  int join_server(int port_number);
98  /* キーボード入力されたメッセージを送信する */
99  void send_msg_from_keyboard(int sock, char *p_buf);
100
101 // ============================================
102 //  linear_lists.c
103 // ============================================
104 /* 線形リストの先頭を取得する */
105 member_t get_head_from_list();
106 /* 線形リストにユーザを追加する */
107 void add_user_to_list(char *username, int sock);
108 /* 線形リストからユーザを削除する */
109 void delete_user_from_list(int sock);
110 #endif
```

11　フローチャート

プログラムの主要な部分のフローチャートを以下に示す。

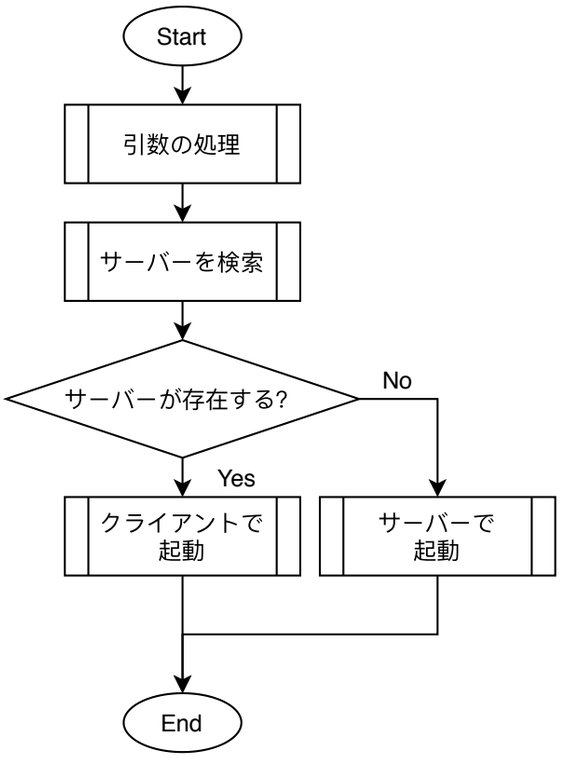

図 1　エントリポイントのフローチャート

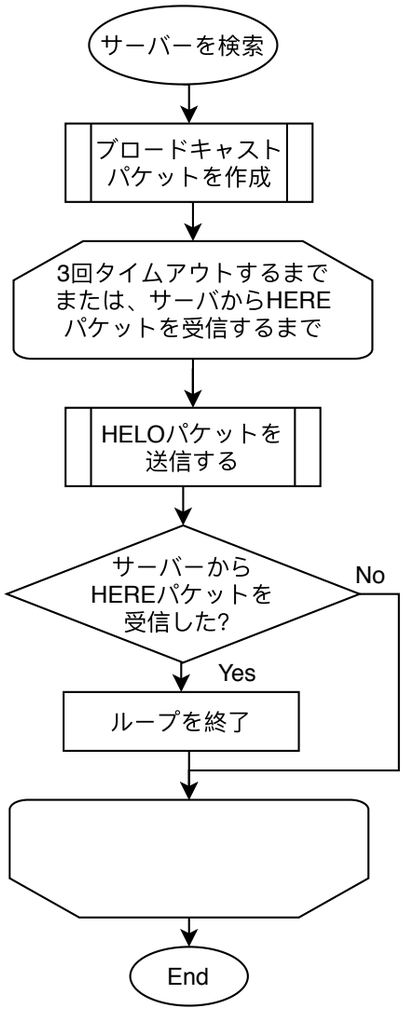

図 2　サーバー検索時のフローチャート

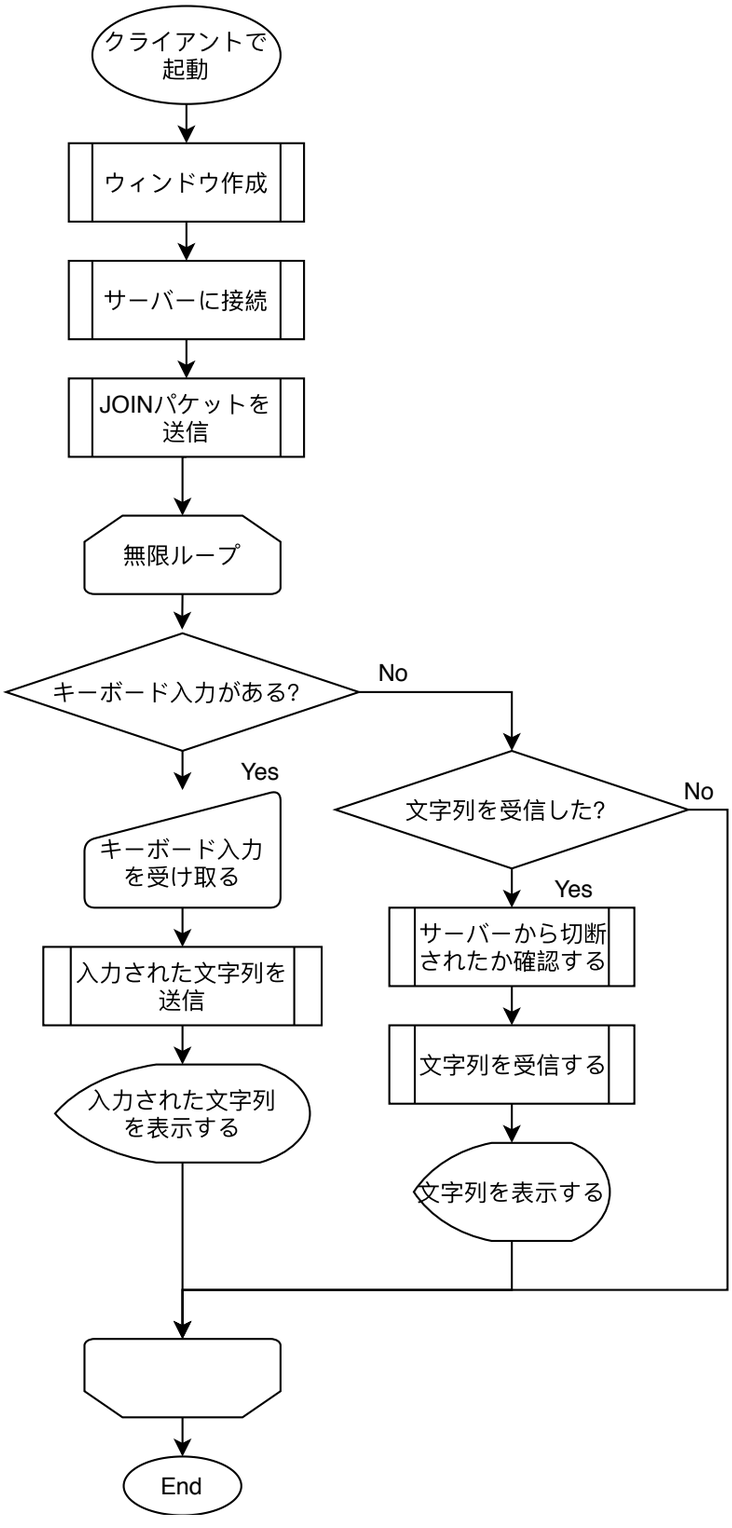

図 3　クライアントのフローチャート

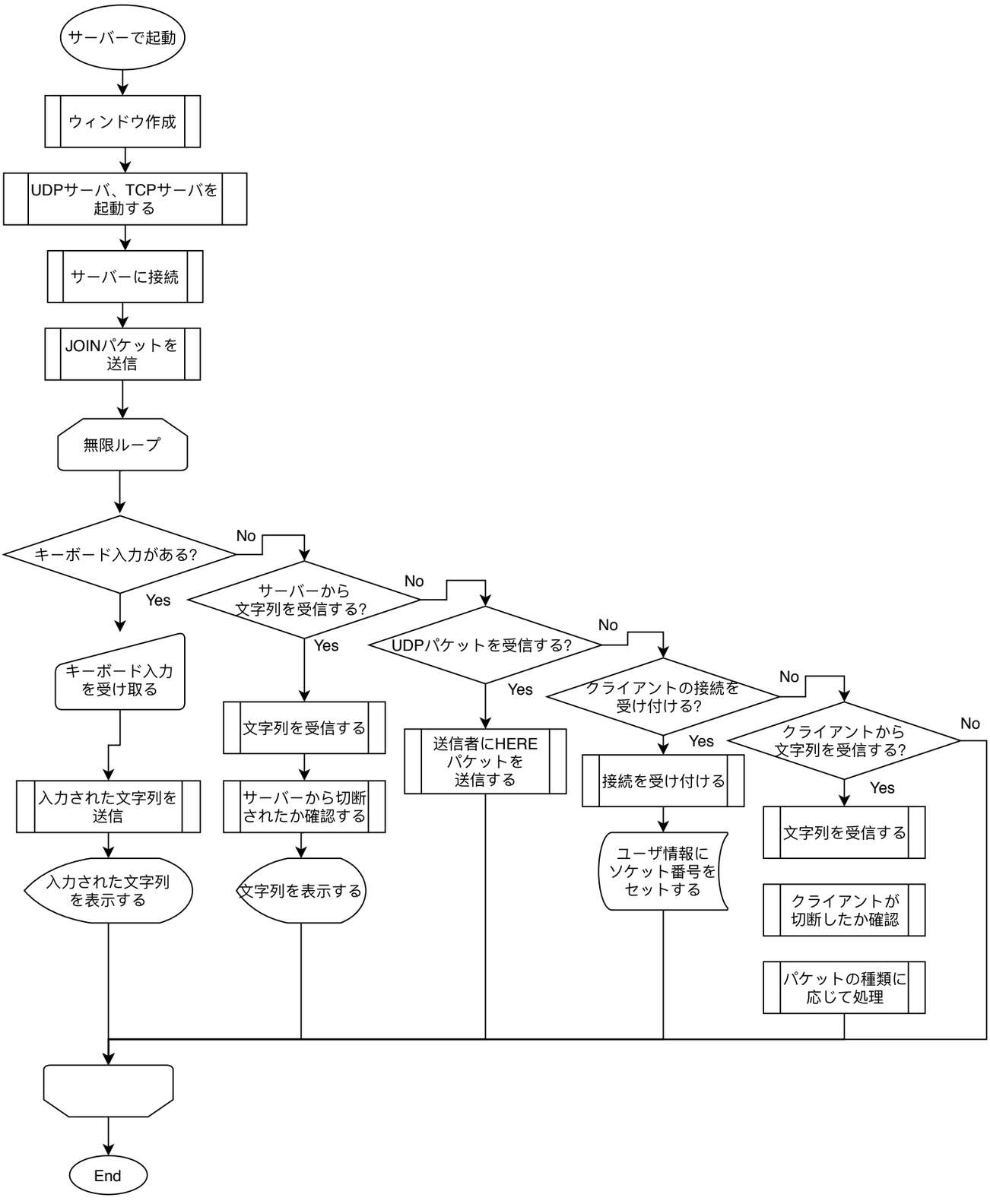

図 4　サーバーのフローチャート

12　クライアントからの不定期なデータ処理

クライアントから不定期にデータが届くため、それに対応する必要がある。そこで、サーバーのメインループ部分は以下のようなプログラムを実装すれば良いと考えられる。

ソースコード 16　idobata_server.c のメインループ

```
/* メインループ */
while (1) {
    /* 受信データの有無をチェック */
    readfds = mask;
    select(Max_sd + 1, &readfds, NULL, NULL, NULL);
    /* キーボードからの入力があった時 */
    if (FD_ISSET(0, &readfds)) {
        char p_buf[MSGBUF_SIZE];
        /* キーボードから文字列を入力する */
        wgetnstr(win_sub, p_buf, MSGDATA_SIZE - 2);
        /* 入力した文字列をサーバーに送信する */
        send_msg_from_keyboard(client_tcp_sock, p_buf);
        /* 自分が入力した文字列を表示する */
        show_your_msg(win_main, p_buf);
    }
    /* UDPパケットを受け取る時->サーバーを探しているクライアントがいた時 */
    else if (FD_ISSET(server_udp_sock, &readfds)) {
        /* 送信してきたクライアントに対して、HELOパケットを送信する */
        recv_udp_packet(server_udp_sock);
    }
    // 新しいクライアントがコネクション張ろうとしている
    else if (FD_ISSET(server_tcp_sock, &readfds)) {
        /* クライアントの接続を受け付ける */
        int sock_accepted = Accept(server_tcp_sock, NULL, NULL);
        /* 名前はJOINパケット受信時に登録するので、ここでは設定せずに登録 */
        add_user_to_list("", sock_accepted);
        /* クライアントとの通信を確認するため、ビットマスクをセット */
        FD_SET(sock_accepted, &mask);
        /* selectで監視する最大値を更新 */
        setMax_sd(sock_accepted);
    }
    /* サーバーからメッセージを受け取った時 */
    else if (FD_ISSET(client_tcp_sock, &readfds)) {
        char r_buf[MSGBUF_SIZE];
        /* メッセージを受信する */
        int strsize = Recv(client_tcp_sock, r_buf, MSGBUF_SIZE - 1, 0);
        if (strsize == 0) {
            wprintw(win_main, "井戸端サーバーから切断しました。\nキー入力でクライアントを終了します。\n");
            wrefresh(win_main);
            close(client_tcp_sock);
            /* 何かのキー入力を待つ */
            wgetch(win_sub);
            return;
        }
        r_buf[strsize] = '\0';
        /* 受信したメッセージを表示する */
        show_others_msg(win_main, r_buf);
    }
    /* サーバーがクライアントからメッセージを受け取った時 */
    else {
        /* そのクライアントからメッセージを受信し、他のクライアントに転送する */
        recv_msg_from_client();
    }
}
```

どのクライアントからメッセージを受信すればよいのか、パケットの種類に応じた分岐処理はどうすればよいのかについて、以下にプログラム例を示す。以下では、`static void recv_msg_from_client()` という関数として定義している。

ソースコード 17　recv_msg_from_client の実装

```
static void recv_msg_from_client() {
    char r_buf[MSGBUF_SIZE];
    int strsize;

    // 線形リストの先頭を取得する
    member_t current = get_head_from_list();
    // 線形リストを走査し、全メンバーのsockを確認する
    while (current != NULL) {
        int client_sock = current->sock;
        ido_packet_t *packet;
        /* selectを用いてメッセージが届いているか確認する */
        if (!FD_ISSET(client_sock, &readfds)) {
            /* このユーザからはメッセージが届いていないのでスキップ */
            current = current->next;
            continue;
        }
        // このユーザから受信する
        strsize = Recv(client_sock, r_buf, MSGBUF_SIZE - 1, 0);
        r_buf[strsize] = '\0';
        packet = (ido_packet_t *)r_buf;

        // 切断された時
        if (strsize == 0) {
            delete_user(current->username, current->sock);
            current = current->next;
            continue;
        }

        // 名前が未設定->まだ参加していない
        if (strlen(current->username) == 0) {
            /* 名前を登録する */
```

```
32              /* なお、そのパケットがJOINパケットのときのみ、名前登録が行われる */
33              register_username(current, packet);
34              current = current->next;
35              continue;
36          }
37
38          // ヘッダで分岐させる
39          // ここに来るメッセージはPOSTかQUITしか無い
40          switch (analyze_header(packet->header)) {
41              case POST:
42                  // メッセージを転送する
43                  transfer_message(packet->data, current->username, current->sock);
44                  break;
45              case QUIT:
46                  // ユーザの登録情報を削除する
47                  delete_user(current->username, current->sock);
48                  break;
49              default:
50                  break;
51          }
52          current = current->next;
53      }
54  }
```